(12) (18) (9)
ARI TECHNICAL REPORT.
(19) TR-78-A8

(6)
A Description of Basic Author Aids in an Organized System for Computer Assisted Instruction.

LEVEL II

by
(10) Roy Kaplow
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Contracted by:
BATTELLE COLUMBUS LABORATORIES
Columbus, Ohio

(12) 37p.

(11) SEPTEMBER 1978

(15) Contract DAJC04-72-A-0001

Beatrice J. Farr, Project Scientist
Leon H. Nawrocki, Work Unit Leader
Educational Technology and Training Simulation Technical Area, ARI

Prepared for

(16) 2Q763731A762

U.S. ARMY RESEARCH INSTITUTE
for the BEHAVIORAL and SOCIAL SCIENCES
5001 Eisenhower Avenue
Alexandria, Virginia 22333

407 080 LB

U. S. ARMY RESEARCH INSTITUTE
FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency under the Jurisdiction of the Deputy Chief of Staff for Personnel

JOSEPH ZEIDNER
Technical Director

WILLIAM L. HAUSER
Colonel, US Army
Commander

Research accomplished
under contract to the Department of the Army

Battelle Columbus Laboratories

NOTICES

DISTRIBUTION: Primary distribution of this report has been made by ARI. Please address correspondence concerning distribution of reports to: U. S. Army Research Institute for the Behavioral and Social Sciences, ATTN: PERI-P, 5001 Eisenhower Avenue, Alexandria, Virginia 22333.

FINAL DISPOSITION: This report may be destroyed when it is no longer needed. Please do not return it to the U. S. Army Research Institute for the Behavioral and Social Sciences.

NOTE: The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR-78-A8	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A DESCRIPTION OF BASIC AUTHOR AIDS IN AN ORGANIZED SYSTEM FOR COMPUTER ASSISTED INSTRUCTION		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Roy Kaplow		8. CONTRACT OR GRANT NUMBER(s) DAJC04-72-A-0001 (Task Order 74-424)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Battelle Columbus Laboratories Columbus, Ohio		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 2Q763731A762
11. CONTROLLING OFFICE NAME AND ADDRESS Office of the Deputy Chief of Staff for Personnel Washington, DC 20310		12. REPORT DATE September 1978 13. NUMBER OF PAGES 27
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Army Research Institute for the Behavioral and Social Sciences, 5001 Eisenhower Avenue, Alexandria, VA 22333		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Research monitored technically by Dr. Leon H. Nawrocki and Dr. Beatrice J. Farr, Educational Technology & Simulation Technical Area, ARI.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Assisted Instruction
programming languages
Instructional Systems
Training computer

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is one of a series of papers dealing with the authoring process and related problems in computer based instruction (CBI). It describes the design of a system of authoring, including some aspects of the actual programming language. The paper provides details on a number of author aids that can be implemented in any organized system for CAI. The topics discussed are:

1. Framework for structuring Author Programs

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (continued)

2. Language Characteristics
3. Display-of-Program Tools and Program Documentation
4. "Built-In" Operations and Subroutines
5. Controls for the Authoring Environment
6. System Maintenance of the Program Data Base
7. Creating Data Bases for Student Use
8. Automation of Student Run Time Facilities
9. Preliminary - Trial Tools
10. Editing Facilities
11. Training of Authors

A major premise of this paper is that the only system which can maximize author assistance is one which is organized along those lines from its inception. Experinece has demonstrated that it is far less satisfactory to tack on author aids to an existing programming language.

CONTENTS

FIGURES

FOREWORD

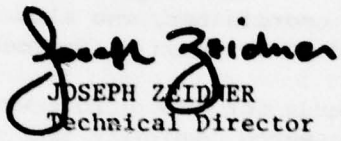
This work was conducted as part of the Army Research Institute's (ARI) research effort on the application of computers in education and training. The work was initiated and funded during FY 75 within the Unit Training and Educational Technology Technical Area under the direction and guidance of Dr. Frank J. Harris, Chief and Dr. Joseph S. Ward, Work Unit Leader. During FY 76 the Educational Technology and Training Simulation Technical Area assumed the task of completing and documenting this work. Acknowledgement is extended to Dr. Beatrice J. Farr, the conference coordinator, who also served as editor for all technical papers, and to Dr. Leon H. Nawrocki, who chaired the sessions.

The primary impetus for this effort was a prevailing feeling among members of the DoD research community that there was insufficient interaction between individuals engaged in research, those involved in developing authoring languages and creating software, and purveyors of hardware. As a result, system requirements for users (authors) were frequently neglected to the detriment of systems effectiveness. To partially correct this situation, ARI conducted a three day conference of selected representatives from each of these domains to discuss mutual interests and problems. The conference was directed toward facilitating research directions necessary for the effective application of computers to training needs, with military training as a focal point.

Through the Scientific Services Program of the US Army Research Office, a contract was let with Battelle Columbus Laboratories to procure the services of ten scientists and educators currently involved in a wide variety of endeavors relating to CAI. These experts, along with ARI staff members, and technical and user representatives from each of the services research organizations or operational CAI activities were the primary participants of the conference (see Appendix A). Additionally, more than fifty individuals from DoD and other goverment agencies, private research groups and academia were invited to the first day of the meeting as observers. The conference was held 9-11 September 1974 in Alexandria, Virginia. During the first morning session, representatives from the Army, Navy, and Air Force gave formal presentations detailing past and present activities related to computer-based training. Attention was also focused on current and anticipated problem areas. The afternoon consisted of exchanges between the participants and observers, and the remaining two days were spent in small group problem solving sessions among participants followed by summary group presentations.

As initially envisioned, the working sessions were expected to emphasize the authoring process. Although the major focus remained as planned, during the course of the conference it became clear that the scope of the problem necessitated examination of artificial intelligence, networking and models describing students, instructors and the learning process.

The goal of the workshop was to document the consensus of this diversified group of experts with respect to: defining user needs and requirements for author languages, identifying deficiencies within existing languages and establishing priorities for future research. Although participants did identify a number of the most critical issues, divergent views emerged regarding research approaches directed toward these issues. Consideration was given to the relative merits of student autonomy, system control or mixed initiative systems. A variety of specific applications were considered and the special problems of authoring in student-controlled instructional environment were explored in depth.



JOSEPH ZEIDNER
Technical Director

A DESCRIPTION OF BASIC AUTHOR AIDS IN ORGANIZED SYSTEM FOR COMPUTER ASSISTED LEARNING

BRIEF

Requirement:

This paper is the first in a series of reports emerging from a conference on research problems and directions for computer based instruction systems. The conference was sponsored and conducted by the Army Research Institute as part of the FY 75 Technology Base Work Program and included in the "DoD Integrated Plan for the Use of Computers in Education and Training."

Approach:

A three day conference was conducted to determine research issues relevant to the improvement of the interface between computer based instructional systems and instructional developers (authors). Participants consisted of ten technical consultants charged with determining and reporting on major topic areas. Additional invited technical and user representatives (governmental, industrial and academic) participated either actively or as observers throughout the conference (Appendix A provides a list of participants). The first day was devoted to (1) formal presentations by military training system representatives describing current and planned computer based instruction activities within the military, and (2) roundtable discussion to delineate and define major topic areas to be addressed. During the following two days participants were divided into 4 working groups. Each group presented a summary of key issues and approaches to authoring system research. Active participants were assigned follow on report topics from these summary items.

Determinations:

A programming system should provide a structural basis capable of assisting authors in organizing their concepts. This structural basis should prossess a multidimensional addressing scheme which facilitates "moving around" in the program as the author works on it, and permit easy reference to specific items which need to be examined or changed. While using only a minimum number of fundamental constructs, an ideal system would allow virtually an infinite variety of effective program structures to be created.

The structure of computer programs is determined principally by the task or interaction which the program is designed to accomplish, but it should also make allowances for the author's personal style (within the constraints imposed by the structure of the particular programming language that is being used).

An author language in the type of system that this report describes must be designed from the point of view that the actions of the language are reflected in operations upon the on-line data base. It should be quite possible to make rather extensive dynamic modifications to the system with minimal author interactions, since required changes in the structure and content of existing data bases can often be done automatically.

Utilization of Findings:

Experience in the Army has shown that the length of time authors (in the sense of courseware developers) will serve in that capacity tends to be two years or less. Hence there is a need to rapidly achieve an acceptable performance base to maximize their availability, and hence cost effective deployment, within the training system.

In the case of computer based instruction, full proficiency (defined as the ability to enter and edity text, develop lessons on or off line, in addition to developing simple macros and sub-routines) should ideally be achieved within six months.

If the preceding need it to be met, specific procedures must be incorporated within the instructional system itself. This report addresses those procedures and, in part, describes the functional requirements necessary to provide the appropriate procedural capabilities.

A DESCRIPTION OF BASIC AUTHOR AIDS IN AN ORGANIZED SYSTEM FOR COMPUTER ASSISTED LEARNING

INTRODUCTION

Readers who are familiar with the TICS system will recognize that much of the framework for this paper is embodied in that system. This is partially because of the role the author has played in the development of TICS. Also, it is a useful example for many of the concepts which we will discuss, since it is one of the few systems for computer-assisted instruction[1] designed to include many of the features now referred to as "author aids." At the same time, it is not my purpose here to describe the TICS system nor to necessarily limit ourselves to features which it provided.

It is contended that in order to maximize assistance for the author, a system for computer-assisted instruction must be organized with that as a principal goal. It will generally not suffice to add a few author aids to a programming language, and it will be difficult to provide a full set of desired capabilities if the underlying system is ill-prepared.

Thus, the present paper, stresses certain fundamental design aspects of a programming system, of which only a small portion is the programming language itself.

These basic features include:

- providing a structured format for the author's program;
- treating the program as a data base, containing not only the usual computer-commands-to-be-executed, but also information about the content and structure of the program;
- separating the operational aspects of the authoring and delivery components of the system and including explicitly the concept of converting the finished program into different forms, suitable for execution on different hardware systems;
- explicitly including the notion that the program, viewed as a structured data base, is not simply a text file but a collection of information--much of which is self-descriptive--and which is organized to be amenable to study and examination;
- discarding the notion that a program must be complete--in any sense--in order for it to be tried meaningfully;

---

[1] It is in keeping with current notions to emphasize the use of the computer as a freely available learning tool, rather than as an additional formal medium of instruction. Nonetheless, I will use the historically entrenched term, computer-assisted instruction (CAI) throughout most of the paper.

- providing mechanisms for the computer itself to automatically perform many of the functions normally associated with programming, such as: checking structural completeness at a local level, finding errors caused by editing associated with program cross-references, generating default behavior for commonly encountered execution-time faults, allowing a large set of student-initiative actions.

The discussion will be separated into a number of parts for convenience in focussing on the particular types of assistance that a system can provide. At the same time, it will be recognized that coherence and internal uniformity within a system is itself an important factor in determining ease of use; the connections and over-laps among the topics will therefore be large in any implementation. The topics will be:

1. Frameworks for Structuring Author Programs
2. Language Characteristics
3. Display-of-Program Tools and Program Documentation
4. "Built-In" Operations and Subroutines
5. Controls for the Authoring Environment
6. System Maintenance of the Program Data Base
7. Creating Data Bases for Student Use
8. Automation of Student Run Time Facilities
9. Preliminary-Trial Tools
10. Editing Facilities

FRAMEWORKS FOR STRUCTURING AUTHOR PROGRAMS

A system for programming should provide a structural basis which helps an author to organize his concepts. Ideally, it should provide structural units which can match the author's conceptual units. Secondly, the structural basis should manifest a multi-dimensional addressing scheme which makes it easy for the author to move around in the program as he works on it, and to refer to specific items which need to be examined or changed. At the same time, the system must allow essentially an infinite variety of effective program structures to be created, preferably with all structures based on the use of a minimal number of fundamental constructs.

All computer programs have a structure to them, determined mainly by the task or interaction which the program is to accomplish and also

by the author's style as well as the structure of the programming language being used. The structure of a completed program can rarely be diagrammed as a one-dimensional list. However, that is the intrinsic form provided by the majority of programming languages in which statements follow statement in a linear address space. It is not surprising therefore that much of the recent discussion of "structured programming has been concerned with the "problem" of branches; visualization of the explicit or conditional flow paths represented by branches requires jumping out of the one-dimensional program space to reach another spot. This is analogous to keeping track of objects which pop into a fourth dimension and reappear elsewhere, and explains why initial program designs or flow diagrams are almost always diagrammed two-dimensionally. Certain types of computer-assisted instruction programs are particularly complicated in this regard, with large numbers of conditional flow paths prescribed, which depend (during execution) on the exact circumstances of the interaction.

It is entirely feasible to provide a two-dimensional program structural basis. We can design a programming system such that programs are constructed out of separate units, called nodes or blocks with each unit being connectable to the others by conditional branches. This is depicted schematically in Figure 1. The issue is not simply to allow such a structure conceptually, but to utilize it as an inherent part of the author environment. Thus the system would include concepts such as:

- at any instant, the authors' statements refer to a "current-working-node;"
- nodes have names, numerical identifiers, and keyword phrases attached to allow ready reference;
- branches are explicitly double ended (e.g. from node A to node B) and can be traced in both directions;
- the address space is multi-dimensional. That is, each item in the program can be referred to in terms of the larger subunit which contains it (if any), its item-type, and its name or sequence location. For example, the author can refer to the "second action (in the list of actions which are to be executed if the) third condition (is true in the) node named sample-name."[2]

There is a second kind of structure to be considered, also, at the level of the programming within each node. This is affected by the programming statements provided, by the extent to which the format is fixed (if at all), and by the actions which are system-implicit for execution when the program is used, without requiring explicit author instructions.

---

[2]The phrases in parentheses in this schematic reference would be implicit and not actually included as part of any real reference to an item.

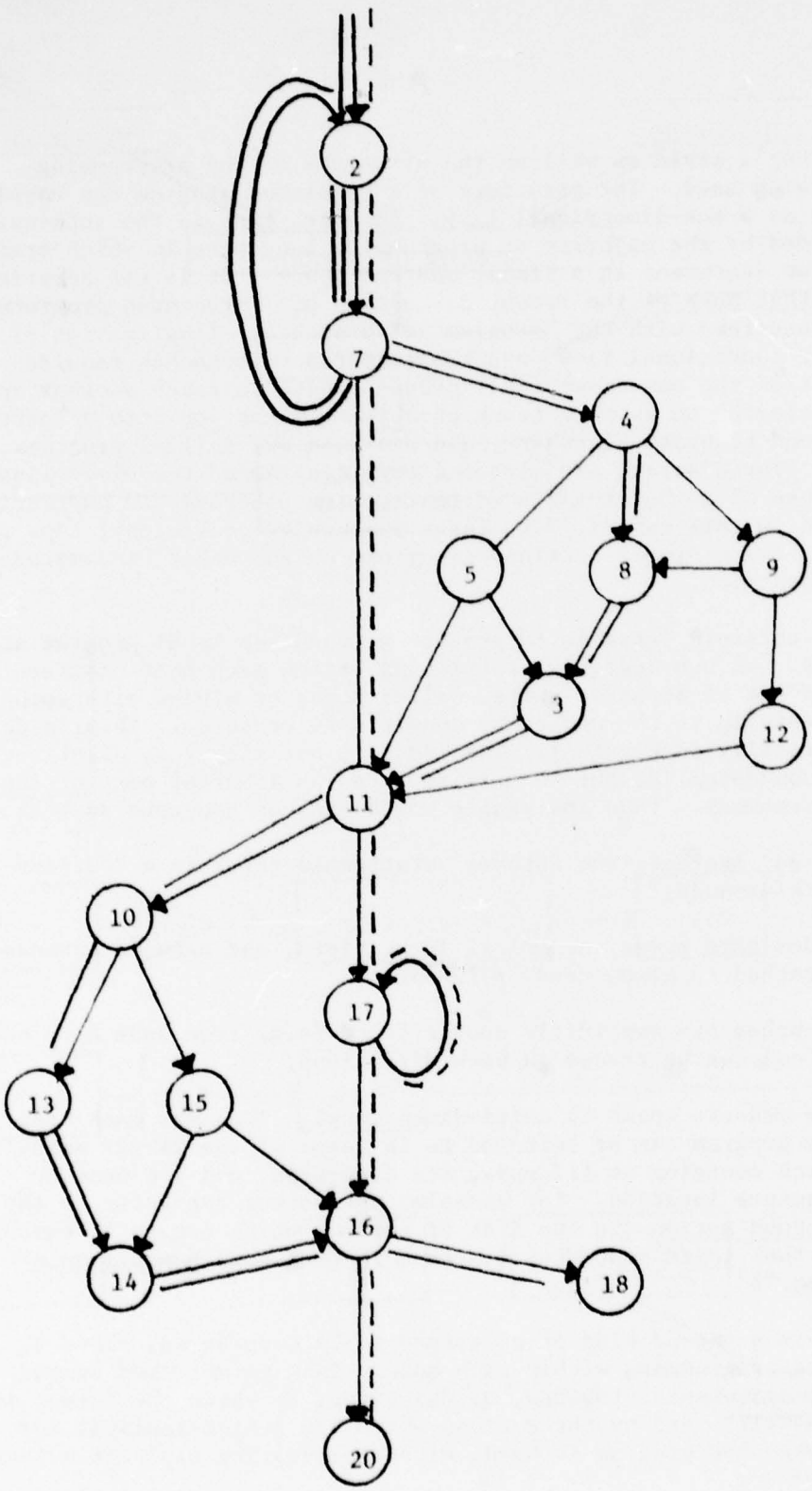


Figure 1. "Net" Diagram of a Portion of a Program, Showing Nodes and Internal Branches. (Two Different Potential Paths are Shown, _____ and --------.)

It is important to point out that there is no need for a node to correspond to the concept of a presentation-frame in an interaction (although it could be used in that way) or for different nodes to have identical or even similar effective internal structures. There is a tendency, when thinking of CAI systems, to imagine a particular, fixed interaction format; in the current context, this translates into providing a template or small set of templates for the internal node structures. If the aim were only to achieve the greatest simplifications of the authoring process, this would be an attractive direction. With a fixed format, the system can provide a maximum of checking and prompting; i.e., the author can be guided through a form-completion process. The apparent advantage, however, is overridden by the fact that examination of what authors actually do when such constraints are not imposed, indicates that the number of templates required is somewhat larger than the number of authors. A more important goal, therefore, is to not constrain unnecessarily the structure of the interaction that will ultimately be executed. (The system designer may nonetheless be tempted to utilize the template concept for novice users, on the basis that it is advantageous for a beginner when all input is explicitly requested. This mode - if provided - should be thought of as a training tool, however, rather than as a permanent author aid.)

To support the structural framework, it is necessary to provide a language, a format, and an implicit structural basis for node construction which allows infinite structural variations within nodes to be created out of a small number of conceptually simple statements. It is also desirable for the system to be able to provide at least a "proof-checking" of the structural integrity of each node.

It is possible, in fact, to work with only one generalized basic programming statement, which has the form:

if ⟨condition⟩ (is true) then (do) ⟨action⟩ and
⟨action⟩ and . . . and ⟨action⟩ .

A node contains a sequence of such statements; each ⟨condition⟩ is examined in turn and its associated string of actions is carried out if it is true. The sequence is generally repeated until a branch-to-another-node action occurs, with inputs being obtained from the user as demanded either by implicit or explicit ⟨actions⟩. Given that the ⟨conditions⟩ can depend on all of the relevant parameters and that the allowed ⟨actions⟩ are sufficiently encompassing, great generality can be achieved. The ⟨conditions⟩, for example, must be able to depend on how student inputs map into anticipated responses, as well as on author-defined and system maintained variables and parameters. Figure 2 shows schematically the internal structure of a node of CAI form which may be constructed on this basis. Here we see an activity of the form: print an output (question), get a response, try to map the response into one of a set of anticipated responses, carry out various arithmetic or output actions depending on the response given and on other parameters, give hints and get new responses, and sooner or later branch to another

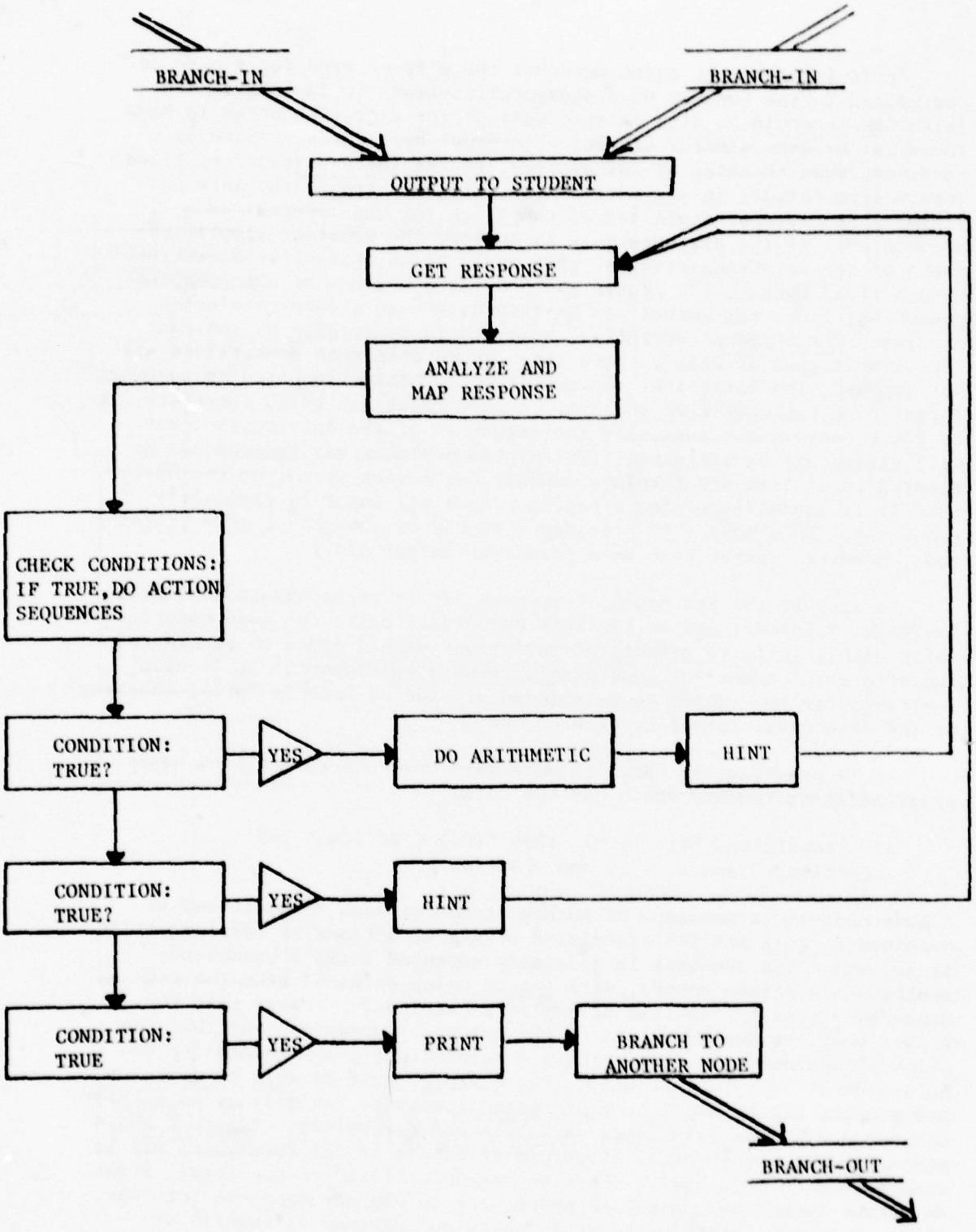


Figure 2. Schematic Diagram of the Internal Structure of a Simple, "Classical" CAI-Type Node.

node. It is not necessary, of course, that all of these things be done in any given node; in fact, a node may be logically complete even if it contains only a single item.

Beyond the elements illustrated here, there are a number of more sophisticated items which can be made available, either for the author's explicit use, or as system-implicit actions. Figure 3 shows a schematic node structure fundamentally like that in Figure 2, but much richer in its potential for student interaction. In addition to the previously shown components, here is added the following:

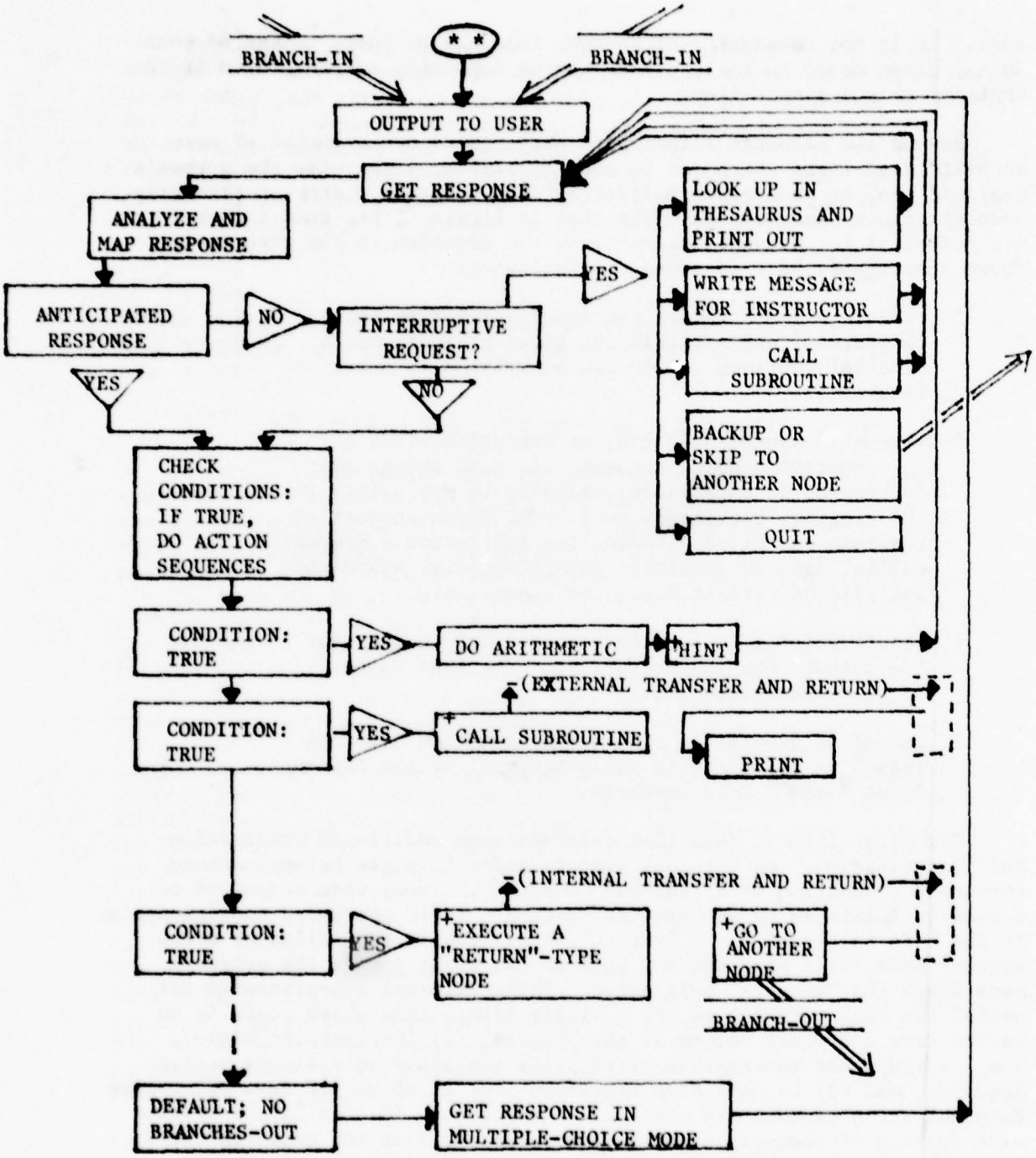
- execute - and - return <actions> applying to both clusters of nodes within the given program and to external routines (which can be written in other languages);

- automatic system handling, at execution-time, of interruptive student requests for such things as: looking-up in data bases, skipping to new points in the program, backing-up to a point where an earlier response was given, sending the instructor a message, calling upon an available subsystem (e.g. numerical analysis or circuit design routines), etc.

- report-writing actions to generate information for the author, the instructor, or - for that matter - for a computerized file.

- automatic generation of a multiple-choice question (based on the author's anticipations) if the student input doesn't lead anywhere.

The first item in this list deserves some additional explanation and justification. An internal execute-and-return can be implemented through the addition of a "return" <action>. Then, when execution of a node is initiated by an "execute" action - as in the third conditional in the node in Figure 3 - a "return", occurring in the called or a subsequent node, will give control back to the point inside the original node where the "execute" originated. These internal sub-processes are useful for such purposes as (1) a single interaction which needs to be called from different points in the program, (2) interactive "hints", i.e., a many-node interaction effectively contained in a single action sequence, and (3) to define an execution-time duration for data allocation. External subroutines serve additional functions: (1) facilitate system-wide sharing of commonly used routines, (2) avoiding the need for duplicating available programming language tools, (3) allowing convenient separation of specialized programming tasks.



(+) These are examples of action types, action sequences can contain any number of actions.

(**) Subroutine calls may be inserted before an output as well as in action sequences.

Figure 3. Schematic Diagram of the Internal Structure of a Node, Including Student-Initiative Options and Transfer-and-Return Actions.

Within this structural framework, including the implicit components, it is also possible for the system to generate, automatically, a useful classification of the structural integrity of each node. Execution of a program must flow from node to node by branches, each of which is specified as an action at the end of an ⟨action⟩ list associated with a ⟨condition⟩. It is, therefore, possible and useful for the system to tell the author, ahead of time, which of the nodes will necessarily branch, which cannot branch under any circumstances, and which will branch only if certain (execution-dependent) conditions are satisfied.

LANGUAGE CHARACTERISTICS

The previous section describes a structural framework which can serve as a base for the programming task. In conjunction with that framework, all the information relative to a program being written is maintained most conveniently as a structured data base, not as a simple text-file listing of program statements. The information which should be maintained includes not just the program itself, i.e., the description of what the computer should do when the program is run, but also auxilliary information that is useful during the authoring process. This includes a lot of rather detailed information, such as back pointers for all branches, cross-references for items developed in one node and used in another, all the locations in which each variable, literal, and subroutine is used, flags which reflect the structural completeness of program subunits, etc.

Thus, the author language in the type of system we are describing must, first of all, be designed from this point-of-view, that is, that its actions are reflected in operations upon the on-line data base. This implies that in using the language the author will be very aware of the structural framework referred to earlier, but the language itself can be quite transparent to the actual structure of the data base and the author need never be concerned at that level. Indeed, it should be possible to make quite extensive dynamic modifications to the system, without troubling authors much, since required changes in the structure and content of existing data bases can often be done automatically.

Secondly, the language must be designed to be the language for a unified authoring system, involving not only writing the program but also examining its logic, creating various displays in its structure, changing it, and trying it out. This point is worth stressing, since the apparently simpler tactic of having separate languages--with different syntax forms and overlapping keywords--for each aspect of the authoring task, is very disadvantageous from the author's point-of-view.

Third, the language itself should be readable. In particular, authors should be able to understand the intent of one another's input so that as a community of workers they can be mutually supportive. This requirement, by the way, is in addition to the need for clear and structured displays or printouts of existing programs or program parts.

Fourth, in regard to author errors, the language needs to be designed to facilitate the system goal of being geared to non-professional programmers, and therefore it should be particularly tolerant of user errors. With respect to the language, this should be reflected first as an effort to minimize the frequency of errors, by choosing syntax and content words that are easy to remember. In addition, the language syntax should allow pinpointing of input errors when they are made, and the system should allow convenient recovery from such errors when they are detected.

Fifth, the language should be convenient to use (which implies all of the foregoing) and also achieve an optimal balance between features that are sometimes antithetical to each other--such as clarity and brevity.

DISPLAY-OF-PROGRAM TOOLS AND PROGRAM DOCUMENTATION

A common belief holds that it is often easier to write a new program than to modify an exisiting one. This attitude arises, of course, from the difficulty normally encountered in following the structure and the logic of a program written by someone else. For two reasons, this is particularly unfortunate in the computer-assisted instructional area: (1) The relevant programs are often very complex structurally, with more path options than are usually found in other types of programs, and (2) The opportunity for continuously modifying, improving, and updating educational material that a computer environment provides is thereby minimized. This latter point deserves some amplification. Teaching, unlike many professional activities, tends not to build on the efforts of predecessors. That is, a teacher usually brings to class (whether lecture, seminar or other format) primarily his own understanding of the subject and his own view of how to present that material. Aside from actual changes which may have occurred in the subject matter itself, for the most part, he is no different from others who have previously taught the subject, nor is there any particular reason why he should do it better. As a result, the art of passing on the most effective teaching "micro-interactions" for a given subject has not been well developed; nor is it clear that it can ever be accomplished, since human interactions are difficult to reproduce, and often awkward and unnatural in reproduction. Similarly, in the writing of textbooks, an author usually takes responsibility for an entire subject area while it may be only a portion of his treatment, that is an improvement over existing work. Indeed, in avoiding the potential charge of plagiarisms, he may select a different, and possibly less cogent, treatment for some parts than was used earlier.

In the new area of computerized instruction, therefore, (or, for that matter, in regard to any medium which is both modular and capable of ready modification) it is desirable to increase the ease with which a person can build upon existing material. Then, after a period of time of ready modification) it is desirable to increase the ease with which a person can build upon existing material. Then, after a period of time

one tends towards refinement of the computer/student interaction, rather than a continuous cycling to new beginnings. (The questions of ownership, copyright, etc., also obviously need to be worked out.) For computer programs, this requires a powerful set of tools through which later authors can first of all understand and subsequently modify and augment existing programs. Since even the initial author will find it difficult to remember the critical details of a complex, interactive program after even a short period of time, such tools will be an important primary authoring aid as well.

In this area as much as others, maintaining the description of the program as a structured data base proves useful. In viewing the structure of a program, the author should be able to request "net" diagrams, such as the one shown in Figure 1. That is, he should be able to ask for the node-branch structure from a given node, forward (or back) a given number of branches. Implicit in such a request is a host of structure-based questions such as: Where do we go from here? How can we get here? Can we get from here to there? With such information in hand, the author will find it useful if he can request a "trace" of a given path. For example, the author should be able to ask to see the outputs and other actions that would occur and the conditions that need to be satisfied for execution of the path through (say) nodes, 4, 5, 3, 11, and 10 (to use identifiers from Figure 1).

For the next level of study, the language should include requests to display in text or (where appropriate) in graphics, the detailed contents of the units of the program, in pieces large or small, utilizing the multi-dimension addressing scheme of the data base. As examples, each of the following might be the object of a display request:

nodes 17, 19, 21-26, 30
node 14
the entire program
the definition of the word "anyword";
or, within the current working node;
the outputs
the third ⟨condition⟩ and its ⟨action⟩ sequence
all anticipated responses
the third anticipated response
the fourth ⟨action⟩ in the second conditional.

In each instance, of course, the printout or pictorial block-diagram should be formatted to reflect the structure of the piece being displayed. In addition, the author should be able to divert the output to a high speed printer or to a high speed, hard-copy, graphical-output device.

A related set of tools is associated with the use of node names and keyword phrases. When properly assigned and with appropriate search requests these allow a detailed and contentful map of the program to be created by the author and used by him and subsequent examiners.

In addition, the system-maintained maps of the use of variables, constants and subroutines, provide another useful mechanism for sensing the program design and constitute an essential aid for making modifications.

Beyond these, the system can also easily include "documentation" features of the ordinary sort, whereby explanatory comments or remarks can be attached to nodes. Indeed, the system can go well beyond the ordinary use of program remarks. It can allow, for example, a listing to be created of only the node identifiers and remarks themselves (and perhaps the keywords); if the remarks are suitably prepared this can provide yet another brief but in-depth view of the program. Also, the system can allow both "active" and "passive" remarks to be attached. The former are intended to act as reminders for the current author, and are printed out each time the node is entered, either as the working node or during a trial-run.

All of these features fit under the general concept documentation, of course. Some apply in the historically limited sense of a printed copy of the program, with explanation. Others enlarge the concept of documentation to include a variety of new modes by which the program can be examined.

BUILT-IN OPERATIONS AND USER SUBROUTINES

Earlier, we referred to a programming statement in which the author specifies which ⟨actions⟩ are to occur if a given ⟨condition⟩ is true. One can think of such ⟨actions⟩ as comprising two categories: those that are explicitly built into the system, and those which are not. ⟨Actions⟩ which are not built-in can be implemented through a single generic ⟨action⟩, perhaps of the form

... call subroutine name (arg1, arg2, ...) ...

if the subroutines are available or can be provided by (or for) the author.

A system can be designed to be sufficiently extensible so that the distinctions between built-in and external operations are not very pronounced. It may be useful, however, to point out the usual distinguishing attributes of built-in operations: (a) the full specification of the action is an inherent part of the language (whereas, for example, the interpretation of the arguments in a subroutine call if done by the external routine itself), (b) the ⟨action⟩ is a guaranteed part of the (student) delivery system - which, in principle, can exist in a variety of hardware manifestations, (c) the ⟨action⟩ is fully documented in the primary user documentation, included in training aids, etc. and (d) the individual built-in ⟨actions⟩ each have their specific place and referencibility in the structured data base description of the program

whereas all subroutine calls are treated alike.

The open-endedness provided by the subroutine call <action> is extremely useful. It makes it unnecessary to duplicate existing programming facilities for writing specialized, efficient, large-scale numerical or other analysis routines. It facilitates splitting-off the associated programming tasks so that they can be functionally described by the author, but actually programmed by another person. It simplifies the sharing of specialized routines and, in general computer utility environment, allows the straightforward incorporation of routines written for other purposes. It also provides a ready mechanism for the system-programmers to effectively expand the "supported" <action> set, without actually having to make ad hoc changes in the language analysis routines, the structure of the data base, or the routines which manipulate the data base.

In the TICS system, there are seven built-in <action> types, which can be described as follows: output text to the terminal and get a response; output text to the terminal; write text entries to a report file; do mathematical or character operations on variables; execute (a cluster of) nodes and return; call an external subroutine, branch to another node. Certain of these also include a number of built-in sub-operations. For example, the output <actions> include facilities for formating and page composition, the mathematical operations include the standard set of numerical functions, and the character operations include the functions normally available in PL/1, e.g. length, concatenate, substring, etc.

On the other hand, the graphical output constructions are provided through two different modes: (a) the use of a number of subroutines, each accomplishing relatively standard tasks such as x-y plots or histograms; and (b) the use of an effectively separate interactive subsystem for constructing drawings, which allows the author to create and store sub-pictures and pictures, which subsequently can be operated on and displayed through subroutine calls during execution of the program. The use of the picture-drawing subsystem, itself a major and independent development project, is illustrative of the flexibility and advantage associated with a general utility operating environment. At the same time, one cannot overlook the fact that to the extent that authors depend on such "external" facilities, the appropriate routines must be provided in new hardware implementations of the entire system or of the (student) delivery system alone.

AUTHOR OPTIONS AND CONTROL OF THE OPERATING ENVIRONMENT

Different authors have different working styles, and different aspects of the authoring job may require different modes of operation. A programming system should be accommodating in these respects.

While on-line use of a programming system is taken for granted, the value of an off-line input mode (e.g. punched card or tape cassette) should not be overlooked. Some authors, for example, will prefer to write parts of the program out on paper, with another person - not necessarily familiar with the system - simply transcribing the input. This operation is often done just as well via off-line input which is usually a less expensive mode. Off-line input can also serve authors whose on-line access to the system is limited by remoteness, terminal availability, or usage and communication costs. It should be recognized that off-line input, in the present context, has the same effect as on-line work, namely to augment, modify, or examine the on-line data base description of the program, and the two modes can be used in juxtaposition.

A related control, to be used on-line, is a block input mode. In this mode, a sequence of instructions can be typed in and subsequently edited if desired, before being released for processing. Some authors appear to prefer this chunk-by-chunk approach during certain parts of the job. For similar reasons, authors want to control the degree of system verbosity associated with the processing of their inputs. Thus, we have found it useful to provide a choice among three levels: verbose, which is fully explicative of the system actions; short, which yields the same information, but in a highly coded, abbreviated form; and, none, in which the system's commentary is essentially limited to error messages.

When working on-line, authors may have different programming strategies which need to be accommodated. For example, one may choose to (try to) fully specify each node in turn, considering all the possibilities that might occur at execution-time in a full sequence of ⟨conditions⟩ and associated ⟨actions⟩. Another may prefer to develop a single path through many nodes, going back to each at a later time to consider other possibilities and their resultant paths. This is a particularly advantageous mode for an author who wants to develop the main line, or "skeleton" of a tutorial first, before putting in remedial material or dealing with specific student errors, etc. It goes without saying that this requires allowing the author to move around freely in his data base, and tools which help him to find his way around to see what needs to be completed; it also means that the author needs to be able to try a program - and to let students use it - while it is structurally incomplete. These tools are discussed in other parts of the paper, especially sections on Automation of Student Run-Time Facilities and Provision for Student Initiatives and Preliminary Trial Tools.

SYSTEM MAINTENANCE OF THE PROGRAM DATA BASE

Reference has already been made to the utility of maintaining the program description as a structured data base during the authoring process. May of the entries made in the data base and operations on it

are implicitly required to allow the system to provide the desired authoring environment. At the same time, we want the system to handle all of the necessary chores with little or no instruction from the author. Examination of some of these will help to clarify what the system must do and the underlying importance of the data base in aiding the author. As each item is created it is assigned a numerial identifier in the multi-level addressing scheme. Interconnections between items are noted; forward and backward pointers are entered for branches; cross-references are entered for items developed in one node and used in another. Tables are kept regarding the locations in which variables, literals, and subroutines are used. Flags are set when modifications to the program lead to possible or certain errors. Subunits are monitored continuously for structural completeness.

Those types of operations are fully automatic. Certain "maintenance" operations require user requests. These include: "garbage collection" and the ordering of items within the data base; checking the stored data for integrity against computer errors; allowing restoration of earlier program states; translation of the program description into a form (or forms) required for one (or more) delivery systems.

CREATING DATA BASES FOR STUDENT USE

Data bases for student use can have a variety of forms and applications. For certain program designs, data bases and the specialized routines which understand and manipulate them are the central aspect of the interaction; for others, these components serve only as an auxilliary.

Structured data bases designed to contain the "knowledge" in a given subject have a particular role in computerized instruction, for the so-called generative mode of operation. Here, ideally, driver programs designed to work with the specific data bases generate questions, check statements, answer questions, etc., in a "conversation" with the student. In the same vein, specialized routines used in conjunction with appropriate data bases can carry out "deep-analysis" of student or author-provided input, to implement a variety of forms of interaction not easily handled in the "anticipated response" framework. Thus, in teaching a foreign language, for example, one can utilize routines which check the structure of student-constructed sentences and answer questions about the structure of sentences presented by the author, and in teaching circuit theory, the computer can calculate numerically quantitative responses and questions about circuits on the basis of similarly programmed models for that subject. This "artificial intelligence" approach is clearly a powerful one, although limited to those topics where the knowledge is sufficiently well organized to allow precise summarization in a computer program and associated data base. So, far, not too much has been accomplished in regard to providing author aids specifically for this purpose, and it is fair to say that professionally competent programmers are needed. Perhaps this is because sufficiently useful generalities about the structure of knowledge have

yet to be developed.

The full subject of data bases in instructional use is much broader, of course. An additional dimension arises in connection with the use of structured data bases of numerical and/or character string information, some of which may be of prior existence. It is clear that for such purposes one wants to allow access to fully general data base structures. Within the TICS system, for example, this is accomplished fairly readily through subroutines which use the data base in question directly and/or which act as interfaces, transferring data between the program and the external data base.

The simplest type of data, normally intended to be used as an auxiliary aid with an interactive program, is just a glossary or dictionary, consisting of an (alphabetical) listing of words or phrases, with each word of phrase attached to a definition. For this, the system can easily provide the author the requisite operations for creating entries and their definitions, and provide both the author and student the requisite look-up and display operations. A useful extension of this concept is to allow each entry to be specified as a thesaurus-type list, a single definition still being associated with the entry. This more realistically recognizes the multiple meanings of words and phrases, which are effectively synonomous. The provided look-up operations must then be sensitive to the additional complication, of course; finding words wherever they appear in thesaurus lists, and dealing appropriately with situations in which words and even partial lists appear more than once. Another extension of the concept is for the system to provide more than one such data base, each with a different name (author selected), to be used in different contexts. In a foreign language program for example, one can be used for the meanings of words, a second for pronunciation guides, and a third for etymological descriptions. As another example, separate data bases can be used in relationship to theoretical concepts, one for definitions, a second to present examples, and a third to explain the relationships among different concepts (and using the thesaurus-list aspect to connect concepts which are related in specific ways). It has also proven to be useful to allow the author to insert a special coded symbol at any point in the definitions, which causes the execution-time system to stop and ask the student if he wants to see more. This gives the data base an inherent "multi-level" aspect.

Such data bases can be useful learning tools for students, even without additional structure being imposed or more complicated data base interrogation routines being provided. An apparently valuable convention is for the author to explain each concept (i.e., define each word or phrase) in terms of whatever words and concepts he believes to be most appropriate, but taking care to similarly explain each concept (or word) he uses which is not to be found in a collegiate-level dictionary. With such a scheme, the structure is implicit in the author's explanations and the student provides his own "tracking" process.

It is often only a matter of cost and convenience whether such data bases are used on- or off-line once they are created. Therefore, a mechanism should be provided for obtaining a formatted and page-composed print-out, suitable for reproduction.

AUTOMATION OF STUDENT RUN-TIME FACILITIES AND PROVISIONS FOR STUDENT INITIATIVES

A system can aid the author through providing automatic run-time facilities which enhance the student interaction without detailed author programming. These can be thought of in terms of two types of activity: first, the implementation of actions which are implicit in the author's instructions and the structural aspects of the system, and second, the provision of options for student initiative and control of the interaction.

Among the first type are the execution-time component of activities which have already been mentioned in respect to the direct authoring facilities. These would include, for example, the response-analysis package; on the one side allowing the author to readily specify response analyses to be done, and on the execution side carrying out detailed response mapping without required detailed author programming. Another example of this type can be seen in the graphical display area where it is again necessary that simple author instructions call into action general purpose and complex routines.

The run-time system should also "back-up" the author by providing sensible default actions, that is, automatic recovery from situations which are logically imperfect during a given interaction. For example, many nodes will be structured such that various branches-out exist, depending on which one of a number of anticipated responses is given. Generally speaking, the interacion defined by a node will be more interesting the larger the number of possibilities covered by the anticipated responses, and the more subtle the differences between them. On the other hand, it is clearly impossible to deal with every conceivable student reponse and request (nor would that necessarily be desirable). The issue, it should be noted, is not simply whether the system/program is prepared to map the arbitrary response into something which is meaningful to it, but also whether it is prepared to carry the interaction into the implied new area. Thus, there need to be mechanisms, author-explicit as well as automatic ones, for dealing with unanticipated student input. At the first level, of course, the author needs to be able to specify the equivalent of:

"if the response is none of those which I have anticipated then,..."; often, the final action specified here will be a "hint" rather than a branch. While the advantage of having locally relevant comments at this

point is great, the ultimate reckoning is only postponed, not eliminated.[3] At the second level, therefore, there needs to be an absolute mechanism for ensuring that the student at least accepts one of the anticipated responses, assuming that he wants the interaction to continue. The obvious mechanism is to create a multiple choice for the student by presenting those responses which the author did anticipate (but excluding those which the student has already given and those which the author prefers to remain hidden). This system-activated default is an important author-aid, especially during the trial-and-development period for a program. In effect, it allows the author to concern himself initially only with the student responses which he believes to be most likely and/or most deserving of recognition. Subsequent analysis of actual trials may then lead to the inclusion of additional anticipated responses and associated sub-interactions.

The concept of student control and initiative during the computerized interaction is important from two perspectives. First, if the system itself is appropriately designed, it is likely that the author's programming task will be easier the more the learning process is left to the student's own judgment. Second, there are a number of pedagogic reasons to believe that the student's learning will be enhanced as he sees himself more in the position of "digging out" ideas which he needs to know, and less as the recipient of a pre-structured flow of information.

Some run-time facilities are closely associated with the notion of student control and initiative but also require author contributions. In this category we include, for example, the student-system portion of data base handling routines that were discussed in an earlier section. Another aspect of student control is exemplified by student-initiated jumps to external routines or to other portions of the given program. Regarding the first, the author probably needs to be able to restrict the extent to which the student can access other routines, expecially in a general purpose utility environment. This can be accomplished, of course, by the author simply including a list of routines which the student should be able to access from his program (and telling the student what those are). Student-initiated jumps to external routines are readily accomplished as "interruptive requests", given at any opportunity for student input. These can be designated by a special code symbol (to distinguish it from an ordinary response), the name of the desired routine, and any parameters (arguments) required. Execution control flows to the routine and back again to the basic program. Normally, the

---

[3] In the TICS system, for example, each conditional-action-sequence which ends with a "hint-and-get-another-response" causes the list of conditions in the node to be reconsidered from the top once again, after the new response is obtained. To avoid "looping", however, individual actions sequences which cause hints are not executed a second time. Thus, even if the author specified a number of conditionals like the one above, each ending with a different hint, a student may ultimately exhaust those explicit interaction components.

request for input would then be repeated. However, in spite of the concept that such "interruptive" jumps to external routines are student initiatives and, therefore, the student's own business, it seems advantageous to "trap" all such returns. This allows the author to monitor the student's external activities, using, at the very least, the values of particular variables in the argument lists. Of course, it is also feasible to create and transfer much more detailed diagnostic information during the student's use of external routines. This type of data would be available for both immediate purposes within the basic program and for subsequent analysis by the instructor.

Similarly, for internal jumps, it will be necessary for the author to restrict the accessibility to a limited number of logical sub-interaction starting points, and also to provide some indication to the student of what those points are and why he might want to jump there. This requirement is assisted, within TICS, by the inherent node-based structure. Thus, the author can attach special keywords (or phrases) to nodes for which he wants to allow "jump-to" access. At run-time, the existence of such a description determines the accessibility of a node and, with appropriate look-up facilities, the list of keyword phrases also gives the student the needed map of the program.

Another type of internal transfer, a "back-up", requires no special author preparation. This student initiative, also implemented as an interruptive request, allows the student to move back to an earlier place in the execution of a program; specifically, to a point where he previously gave one response and now wants to give another. This is the essential student control, for it allows him to explore the optional paths provided, to experiment with answers which he may know to be wrong, and to recover from misinterpreted responses, etc. This capability, when treated as a true roll-back rather than a jump, requires stacking the history of changes in the values of variables, so the original situation can be restored. If a backup over an indefinite length of interaction is allowed, this feature might be very demanding of secondary storage, especially when array variables are frequently changed. Thus, when the student-delivery hardware system is limited, it would probably be necessary to use a "backup-stack" of fixed length, with the maximum backup distance being a dynamic function of variable changes.

An additional student initiative relates to communicating with instructors and the authors of programs. A student should be able to send messages to those persons, while he is using a program. The immediacy of that feedback path, from the student's point of view, enhances its value compared to other means. It is feasible, although by no means necessary, for the message recipient to be on-line, also.

As a last component of facilities which a system should provide to foster student initiative and control, we mention the important area of

cataloging all of the available instructional modules within a system and of allowing students to select freely among them and, indeed, to jump from one to another.

PRELIMINARY TRIAL TOOLS

By preliminary trial tools we mean aids for trying out a program before it is completed, and for obtaining useful information regarding the efficacy of a program when it is in use but still being refined.

A mode of operation can be included in which the author can play the role of a student, inputting appropriately, while the system simulates the execution of the program, starting at any point. This can be done while the program is structurally incomplete and even erroneous. In this mode the system can print out its flow-path, e.g., the node-to-node branching, and the conditionals satisfied and executed. It can detect and report all unsatisfactory conditions encountered during such a trial and when dead-ends are reached, request instructions about whether it should proceed and, if so, from what point. In such a mode, the author can be given a variety of commands for controlling the simulations, and for examining and setting the values of variables. He can also be allowed to set "stop points" at arbitrary points in the program, which cause automatic halting of the simulation, to allow examination of the instantaneous state of affairs. An author can also interrupt such a simulation to examine or to modify any part of the program, and continue the simulation after changes or new entries are made.

It is also useful to try the program with real students, even at very early stages of program development. For this purpose, a mode of simulator operation can also be provided in which the auxiliary (path information) output, the special user-control options, and the mechanisms for directly affecting the program data base are inhibited. Apart from program incompleteness and errors, in this mode the simulation should have exactly the same appearance as the ultimate execution, including for example, the full availability of the student's interruptive requests.

When a structurally complete version of a program is ready, it can be used in the standard delivery system. Usually, however, the author will still be anxious for rapid feedback on the use of the program to further improve the interaction on the basis of broader experience. For this purpose there are at least three usefully automated mechanisms which can supplement direct communication with the student-users. The system can record the history of each student's interaction, that is, of the flow-path, including all of the student's inputs. This allows both a further review of the logical consistency of a program and also a monitoring of the actual student responses (or inputs) given at each point. We have earlier mentioned providing a student facility for sending messages to the author of the program. This is particularly useful when the student encounters errors in a new program. A third mechanism gives the author the ability to "write a report" to himself

during the execution of a program. Specific entries can be written i the report whenever specific conditions obtain at any point in the program. In this fashion, the author can generate a file of whatever data he believes will be useful. This may include, for example, unanticipated responses, notice of particular branches taken, values of variables (or the point at which a given counter reaches a certain value), the time it takes for a student to respond in particular nodes, etc.

When it is desired, it is feasible for a group of such reports to be processed by provided routines, to obtain summary data. In the same vein, it is possible to provide program-specific but student-global variables in which execution-statistics relevant to the entire student user group are continuously maintained. Needless to say, all of these mechanisms are also valuable to non-author instructors, who later happen to have responsibility for the student's use of the program.

EDITING FACILITIES

Many of the features described in previous sections, particularly the ones on Display-of-Program Tools and Program Documentation and Preliminary Trial Tools, relate to examining a program in order to modify it. The process of modifying a program on-line is generally referred to as "editing" since programs are usually line-organized text files, and changes are accomplished with the aid of a text-editor program. In our present context, in which the program description is considered to be a structured data base, the process of modifying the program can benefit from system-provided author aids more closely related to the strucutre of the program and to the types of items contained.

To begin with, of course, the editing operations can all make use of the detailed address referencing scheme inherent in the data base. Thus, the author always works on explicitly designated items (e.g., "the fourth anticipated response") and is not concerned, as a primary matter, with moving a line pointer around in a linear file. A general text editor is still useful, but mainly for those items which are explictly text and which are sufficiently long that it is preferable to modify an existing version rather than replace it. Such items may include, for example, output texts, long anticipated responses, remark entries, dictionary definitions, etc. In addition, a variety of more specific operations for modifying a program can be supplied; a few examples will be given here for illustration. A "delete" operation, for example, can apply to any items in the program data base at different levels of the structure; e.g., delete a specific action in a specific conditional action sequence, delete a complete conditional actions sequence, or delete an entire node. (A deleted item should not really be erased, of course, especially if it is large, since deletions can often introduce major logical errors. Fortunately, the system can immediately detect

such occurrences - as elaborated later - so it is convenient for the author, and worth the overhead, if the option to "restore" the item is retained.) A "move" operation is available to rearrange existing items where order is important, e.g., the order among conditionals in a node, or the order among actions in a conditional sequence. This operation, of course, not only moves a designated item to a specific spot in a list, but automatically "renumbers" the ones that need to be adjusted. A related operation is "insert", applying to the the creation of a new item; again, space is made for the new item and the others are rearranged. A "change" operation allows a specific item to be replaced by a new one of the same sort; e.g., "change the third condition to..." . Still more specialized editing requests are appropriate for certain portions of the program, such as the keyword-phrase list and student data bases (e.g., dictionary/thesaurus).

Recalling that the system is keeping track of many structural details about the program, such as where individual variables are used, it should be noted that it must automatically maintain the integrity of such data through all such author-requested modifications, without requiring any further explicit instructions.

Even given the aids so far described, the editing of a program can be an uncertain task. As implied earlier, the most difficult part of the job for the original author and for subsequent modifiers of the program, is to keep track of the interrelationships among different parts of the program. Often, a change made in one place will have ramifications elsewhere. Fortunately, the system can itself keep track of all of the explicit cross-references among items, both intra- and inter-node. These tables can be examined by the author, of course, prior to his making any changes. More importantly, the system itself can monitor the cross-references when changes are requested, inform the author of potential or certain errors which are thereby introduced, and leave warnings or error message flags attached to the affected items in the program.

TRAINING OF AUTHORS

The training of authors might be construed, in the present context, as a concern solely with teaching prospective authors the mechanisms for using the systems in question. This would include, of course, the language, the auxiliary capabilities, the purpose and use of each command, etc. I believe that if the system is well designed and the language constructs chosen with care, it will not be difficult for an author to learn the rules and to construct and manipulate an arbitrary program using the full facilities of the system as described above. Beyond the usual sorts of printed documentation and explanations, one should obviously consider providing a computerized instructional program to teach the use of the system itself. A particularly interesting possibility exists with the simulation mode described earlier; it is possible to write a program intended to be used in that mode, and which the prospective

author would begin to use in the real-student option. However, he would have the ability to switch, when instructed, to the author mode, where he would be shown how to examine and modify the very program which he was then using. In fact, the transition from student to author could be virtually imperceptible.

Training authors to make optimal use of a system is a broader question, however. It is dependent, in part, on a deeper conceptualization of programming, of structural and interaction units, and of such issues as the balance of student and program initiative. It also depends on pedagogical attitudes and theories, and - certainly not least - on the clarity with which one sees the structure of the subject material and how it should best be presented (or made available) in the computer medium. These questions are themselves deserving of more extensive treatment, but the topics go beyond the boundaries of the present paper.

BIBLIOGRAPHY

Brown, J. S., Burton, R. R., and Bell, A. G., "SOPHIE: A Sophisticated Instructional Environment for Teaching Electronic Trouble-shooting (An Example of AI in CAI)" BBN Report No. 2790, Bolt, Beranek and Newman, Inc., Cambridge, MA. March 1974.

Feurzeig, W. and G. Lukas. "The Use of Dribble Files as Instructional Aids." Bolt, Beranek and Newman, Inc., 1975.

Goheen, S. and D. Jordan, "Evaluation of TICS: A Multics Subsystem for the Development and Use of CAI Courseware." MITRE Technical Report 2749. June 1974.

Hewitt, C. E. and Smith, B. "Towards a Programming Apprentice," IEEE Transactions on Software Engineering. March 1975.

Kaplow, R., et al. "Teacher-Interactive Computer System: I. The Author-Language and Instruction Manual; II. Language Specifications," Massachusetts Institute of Technology. 1971; (updated 1973, 1974, 1975)

Kaplow, R., Schneider, D., Smith, F. C., Jr.,Stensrud, W. R., "Computer Assistance for Writing Interactive Programs: TICS," Proceedings, Association for Computing Machinery, August, 1973.

Kaplow, R., Desch, S. H., Jr., Pettijohn, D. O., Rodman, M. H. and Smith, F. C.,Jr., "Illustrations of Conversational, Inquiry, Problem-Solving, and Questionnaire Type Interactions within the TICS System," Proceedings, Seventh Annual Princeton Conference on Information Sciences and Systems. March 1973.

Levine, D. R. "Computer-Based Analytic Grading for German Grammar Instruction," PhD Thesis, Stanford University, 1973; Institute for Mathematical Studis in the Social Sciences. Technical Report No. 199. March 1973.

Nelson, G. E., Ward, J. R., Desch, S. H., and Kaplow, R. "Two New Strategies for Computer-Assisted Language Instruction, Foreign Language Annals," Foreign Language Annals, 9, 1, 1976.

Shortliffe, E. H., "MYCIN: A Rule-Based Computer Program for Advising Physicians Regarding Antimicrobiol Therapy Selection" Doctoral Dissertation, Stanford University, October 1974. Stanford A.I. Memo 251. Stanford.

Soloway, E. and Riseman, E., "Common-Sense Theory Formation Toward Understanding Baseball," University of Massachusetts. COINS Technical Report 75C-5, 1975, Amherst.

Sussman, G. J. and Stallman, R. M., "Heuristic Techniques in Computer Aided Circuit Analysis," M.I.T. A.I. Lab Memo 328, March 1975. Cambridge

APPENDIX A

PARTICIPANTS

Mr. Avron Barr
Institute for Mathematical Studies in the Social Sciences
Stanford University, Ventura Hall
Palo Alto, CA 94305

Dr. Alfred Bork
Department of Physics
University of California
Irvine, CA 92664

Dr. John Brackett
SofTech
460 Totten Pond Road
Waltham, MA 02154

Dr. Victor C. Bunderson
Institute for Computer Uses in Education
Brigham Young University
Provo, UT 84601

Mr. Frank Dare
CAI Project
USA Ordnance School and Center
Aberdeen, MD 21005

Mr. Wallace Feurzeig
Bolt, Beranek and Newman
50 Mouton Street
Cambridge, MA 02138

Dr. Dexter Fletcher
Navy Personnel Research & Development Center
San Diego, CA 92152

Mr. Ed Gardner
Air Force Human Resources Laboratory
Lowry AFB, CO 80230

Dr. Roy Kaplow
Massachusetts Institute of Technology
Room 13-5106
Cambridge, MA 02139

Mr. Don Kimberlin
Office of Project Manager
Computerized Training System
Ft Monmouth, NJ

Mr. George Lahey
Navy Personnel Research & Development Center
San Diego, CA 92152

Mr. Hal Peters
Hewlett-Packard
11000 Wolf Road
Cupertino, CA 95014

Dr. Mortenza A. Rahini
Department of Computer Sciences
Michigan State University
East Lansing, MI 48823

Dr. Martin Rockway
Air Force Human Resources Laboratory
Lowry AFB, CO 80230

Dr. Robert Seidel
HumRRO
300 North Washington Street
Alexandria, VA 22314

Mr. Robert H. Simonsen
System Development Technology
Boeing Computer Services
Seattle, WA 98108

Dr. Lawrence Stolurow
Division of Educational Research
State University of New York
Stony Brook, NY 11790

Dr. Paul Tenczar
Computer-Based Educationa Research Laboratory
University of Illinois
Urbana, IL 61801

Dr. Karl Zinn
Center for Research in Learning and Teaching
University of Michigan
109 East Madison Street
Ann Arbor, MI 48104

APPENDIX B

PARTICIPATING ARI STAFF

Mr. James D. Baker
Dr. Beatrice J. Farr
Dr. Frank J. Harris
Dr. Cecil D. Johnson
Dr. Bruce W. Knerr
Ms. Martha Moore
Dr. Leon H. Nawrocki
Dr. Michael H. Strub
Dr. Joseph S. Ward